# 22 | 压测平台：如何解决 GoReplay 动态数据关联？

2021-12-07 高楼

《全链路压测实战30讲》 课程介绍 >

讲述：高楼
时长 12:49 大小 11.74M

你好，我是高楼。

在第 6 讲，我们说过目前主流的流量回放工具都无法轻易解决 session 的问题，所以从系统安全的角度来说，工具需要做对应的改造。

这节课，我们来聊一下 GoReplay 如何通过改造解决回放过程中动态数据关联的问题。

## 关联是什么？

我们可以把关联简单地理解为把服务端返回的某个值，传递给后续的调用使用。我们可以在很多场景用到它。举个例子，我们常见的“Session ID”就是一个典型的需要关联的数

据。它需要在交互过程中标识一个客户端身份，这个身份要在后续的调用中一直存在，否则服务端就不认识这个客户端了。

对每一个性能测试工具来说，关联是应该具备的基本功能，GoReplay 也不例外。

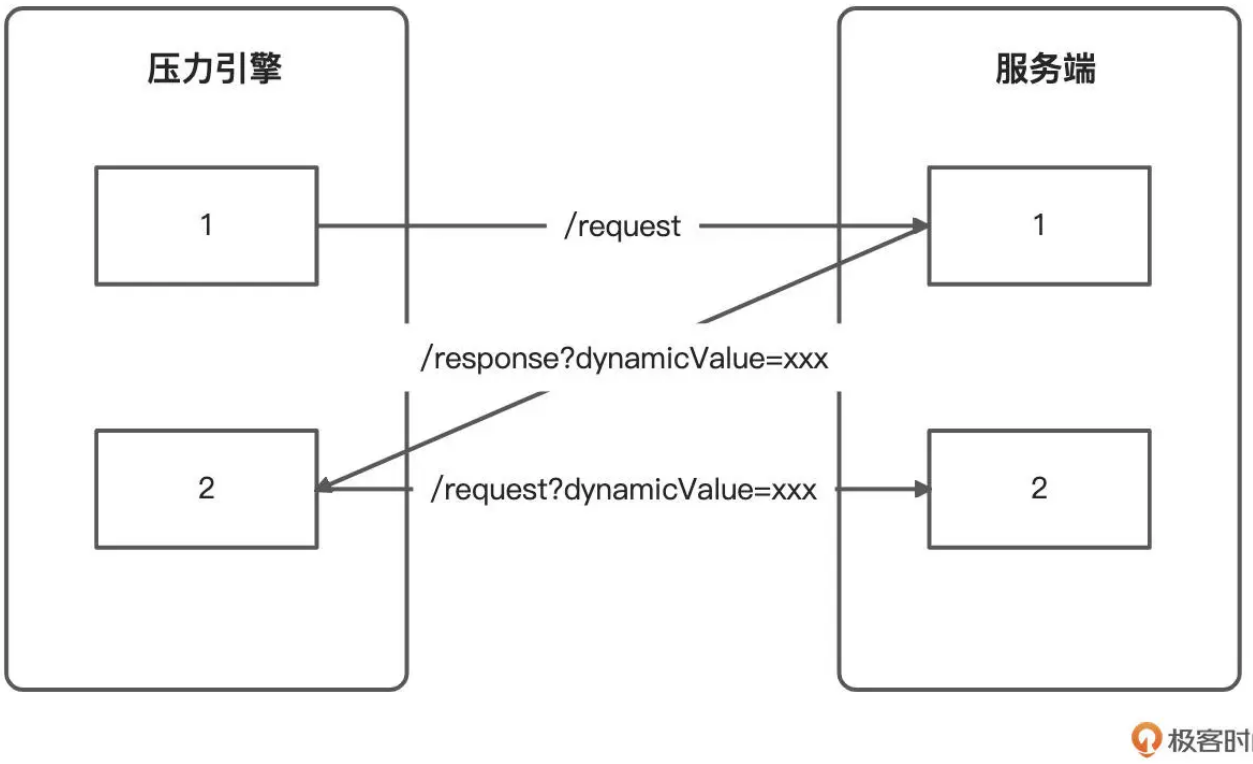
但是有很多新手同学对关联的逻辑并不是十分理解，甚至有人觉得关联和参数化（流量数据）是一样的，因为它们用的都是动态的数据，并且关联过来的数据也可以用到参数化（流量数据）中。其实，这二者还是有所不同的，因为关联的数据后续脚本中会用到，但参数化就不会。

现在有很多全链路压测都是由单接口基准创建的，这样一来，关联就用得比较少。因为接口级的基准场景都是一发一收就结束了，不需要将数据保存下来再发送出去。

那么正常情况下，什么样的场景需要关联呢？一般情况下， 它们需要满足下面几个条件：

1. 数据是由服务器端生成的；
2. 数据在每一次请求时都是动态变化的；
3. 数据在后续的请求中需要再发送出去。

你可以通过这张示意图加深一下理解：




好了，我们知道了关联的基本概念和适用场景，那么在 GoReplay 中又如何改造呢？

作为一款流量回放工具，我们知道 GoReplay 的核心原理就是基于流量文件去倍数回放请求。很显然，这个流量文件是个死的东西，是不能动态参数数据的，那么，我们又该怎么办呢？

这时候，我们就需要搬出 GoReplay 的中间件了。

## 中间件是什么？

中间件（ Middleware ）是一个在 STDIN（标准输入） 接收请求、响应 payload（有效请求负载）并在 STDOUT（标准输出） 发出修改请求的程序。你可以在中间件上实现任何自定义逻辑，比如认证、复杂的重写和筛选请求等。

通过传入 Middleware 参数，我们可以发送命令给 GoReplay，GoReplay 会拉起一个进程执行这个命令。在录制过程中，GoReplay 通过获取进程的 STDIN 和 STDOUT 与输入输出插件进程进行通信，中间件内部逻辑为 STDERR，数据流向大致如下：

复制代码

```
1                    Original request       +---------------+
```

```
+-------------+----------STDIN---------->+              |
|  Gor input  |                           |  Middleware  |
+-------------+----------STDIN---------->+              |
                   Original response     +------+---+---+
                                                |   ^
+--------------+   Modified request             v   |
| Gor output  +<---------STDOUT-----------------+   |
+-----+-------+                                     |
      |                                             |
      |            Replayed response                |
      +-------------------STDIN-------------------->+----+
```

需要注意的是，如果希望记录原始响应和回放响应，不要忘记添加 **– output-http-track-response** 和 **– input-raw-track-response** 参数。

GoReplay 支持用任何语言编写中间件的协议，同时中间件程序还需要格外注意一点，就是中间件和 Gor 的所有通信都是异步，因此，我们不能保证原始请求和响应消息会一个接一个地出现。如果业务逻辑依赖于原始响应或回放响应，那么中间件应用程序就应该处理好状态，也就是要做好动态数据的处理动作。

为了简化中间件的功能实现，官方为 node.js 和 Go (即将推出) 提供了包。

## 如何使用中间件？

那么，应该怎样使用中间件呢？

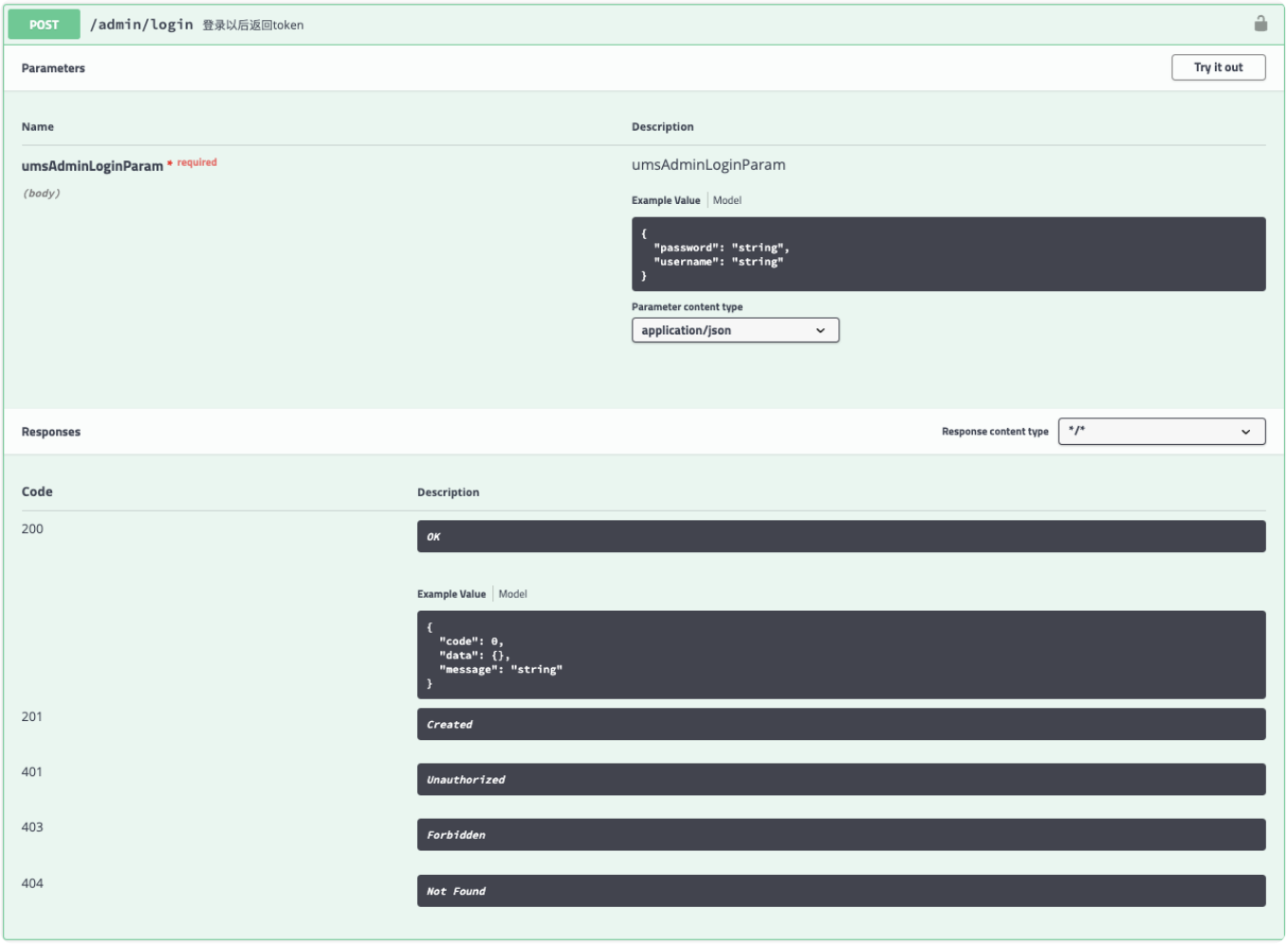
下面就是一个简单的使用 bash echo 中间件的示例，我们用它来打印对应的 payload 类型：

复制代码

```
#!/usr/bin/env bash
#
# `xxd` utility included into vim-common package
# It allow hex decoding/encoding
#
# This example may broke if you request contains `null` string, you may consid
# See: https://github.com/buger/gor/issues/309
#

function log {
    # Logging to stderr, because stdout/stdin used for data transfer
    # 记录到 stderr，因 为 stdout/stdin 用于数据传输
```

```
    >&2 echo "[DEBUG][ECHO] $1"
}

while read line; do
    decoded=$(echo -e "$line" | xxd -r -p)

    header=$(echo -e "$decoded" | head -n +1)
    payload=$(echo -e "$decoded" | tail -n +2)

    encoded=$(echo -e "$header\n$payload" | xxd -p | tr -d "\\n")

    log ""
    log "==================================="

    case ${header:0:1} in
    "1")
        log "Request type: Request"
        ;;
    "2")
        log "Request type: Original Response"
        ;;
    "3")
        log "Request type: Replayed Response"
        ;;
    *)
        log "Unknown request type $header"
    esac
    echo "$encoded"

    log "==================================="

    log "Original data: $line"
    log "Decoded request: $decoded"
    log "Encoded data: $encoded"
done;
```

这里我们使用【会员登录接口】来做演示。

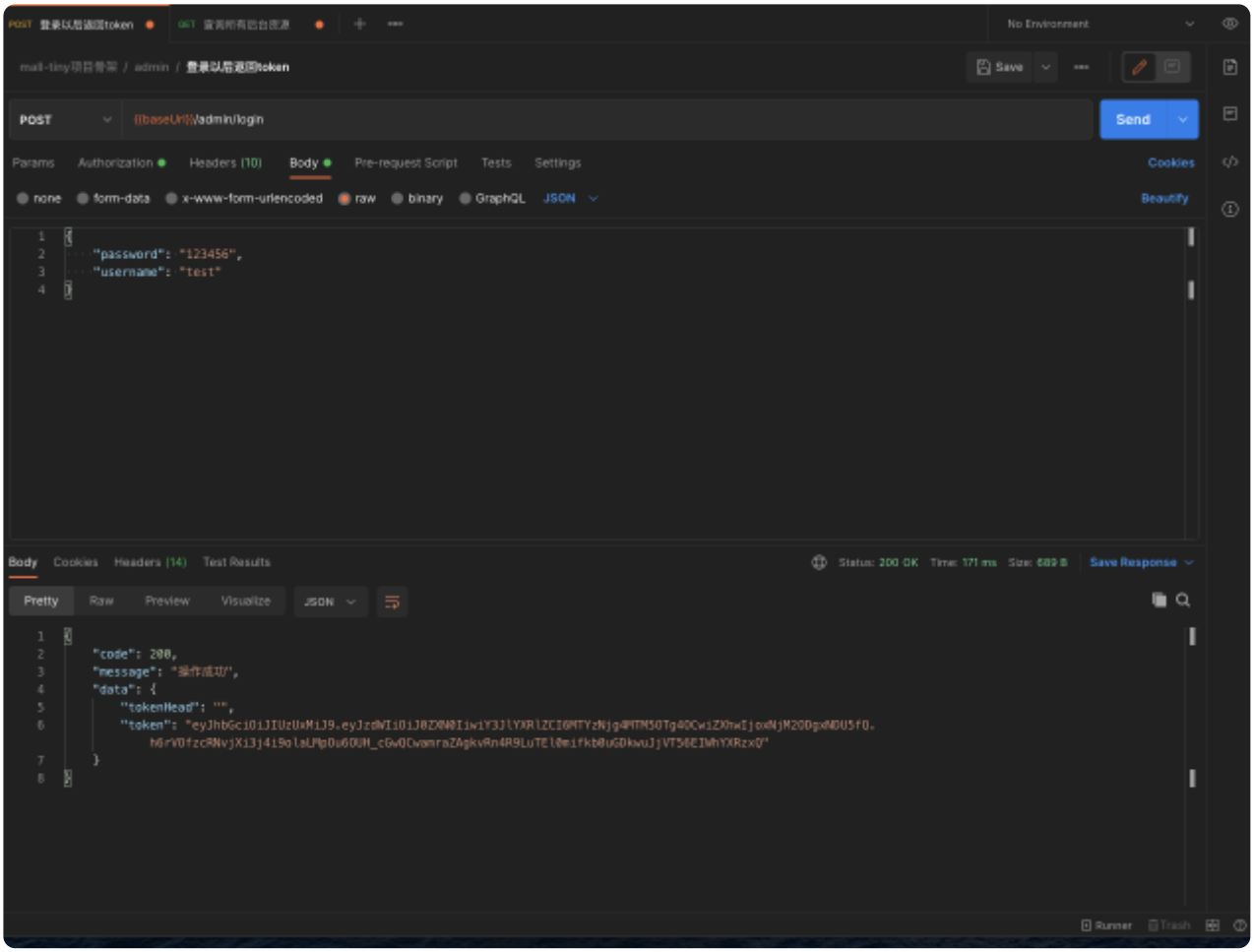



首先，通过指定 Middleware 可执行文件的命令，也就是使用 Middleware 参数在 GoReplay 启用中间件功能：

复制代码

```
1  sudo ./goreplay --input-raw :8081 --middleware "./echo.sh" --output-http "htt
```

接下来，我们通过 Postman 对【会员登录】接口做一次测试。




通过控制台我们看到，中间件程序已经成功把经过的流量信息全部打印出来了。

复制代码

```
Interface: en0 . BPF Filter: ((tcp dst port 8081) and (dst host fe80::8f6:ee40
Interface: awdl0 . BPF Filter: ((tcp dst port 8081) and (dst host fe80::50a5:c
Interface: llw0 . BPF Filter: ((tcp dst port 8081) and (dst host fe80::50a5:ce
Interface: utun0 . BPF Filter: ((tcp dst port 8081) and (dst host fe80::d9a3:a
Interface: utun1 . BPF Filter: ((tcp dst port 8081) and (dst host fe80::c2a0:1
Interface: utun2 . BPF Filter: ((tcp dst port 8081) and (dst host fe80::771:49
Interface: utun3 . BPF Filter: ((tcp dst port 8081) and (dst host fe80::4a93:e
Interface: lo0 . BPF Filter: ((tcp dst port 8081) and (dst host 127.0.0.1 or d
2021/11/14 17:16:34 [PPID 8021 and PID 8022] Version:1.3.0
[DEBUG][ECHO]
[DEBUG][ECHO] ====================================
[DEBUG][ECHO] Request type: Request
[DEBUG][ECHO] ====================================
[DEBUG][ECHO] Original data: 3120636239663166393130303030303030313535353939393
[DEBUG][ECHO] Decoded request: 1 cb9f1f910000000155599937 1636881399760637000
POST /admin/login HTTP/1.1
Content-Type: application/json
Authorization: true
User-Agent: PostmanRuntime/7.28.4
Accept: */*
Postman-Token: 4fa25e6b-fd46-4e96-8af6-f6c5ba0f003f
Host: localhost:8081
```

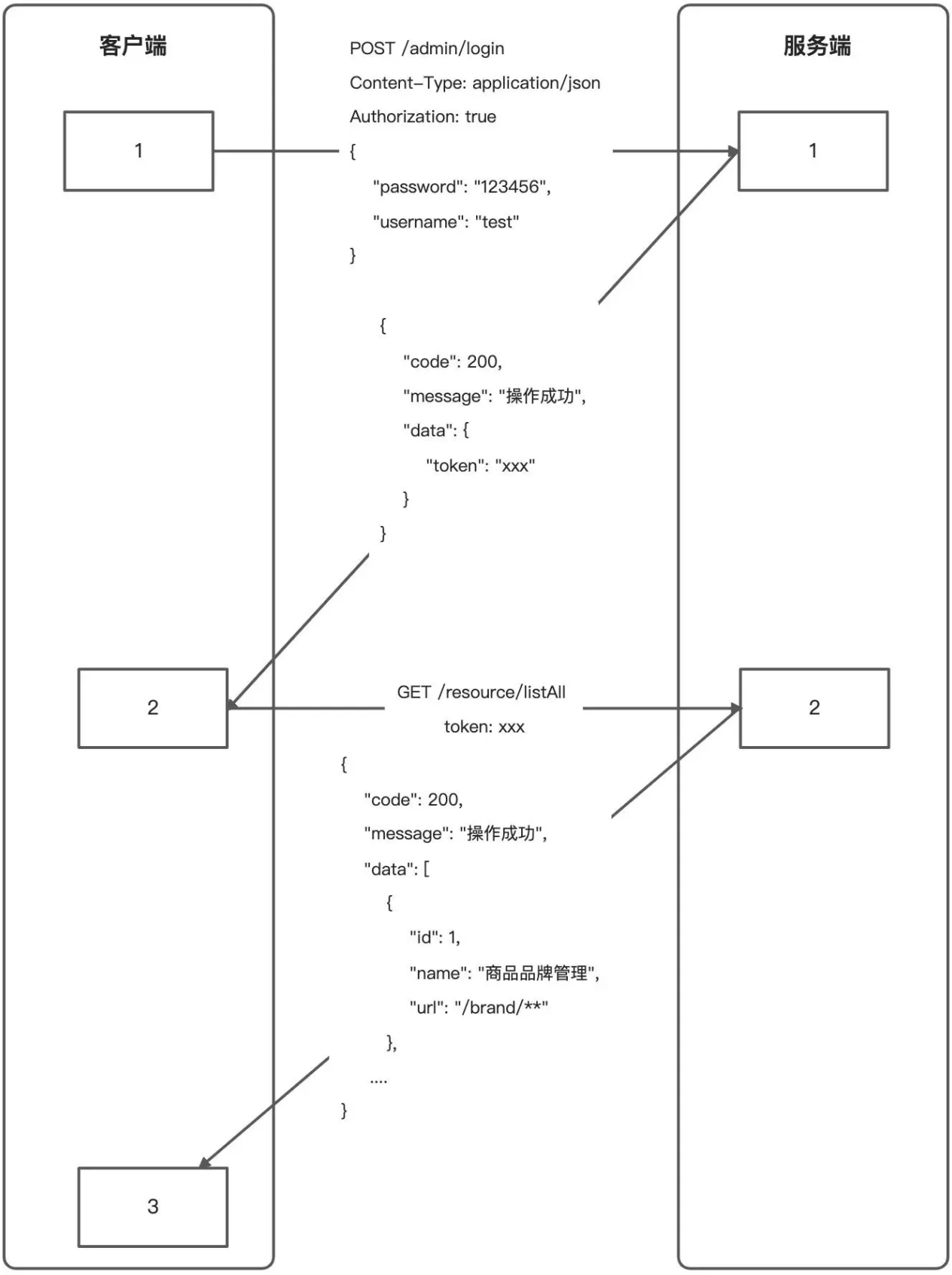
```
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 52

{
    "password": "123456",
    "username": "test"
}
[DEBUG][ECHO] Encoded data: 3120636239663166393130303030303030313535353939393933
```

到这里，我们已经了解了中间件的基本功能和使用方法，接下来我们回到这节课的主题，如何实现关联操作？

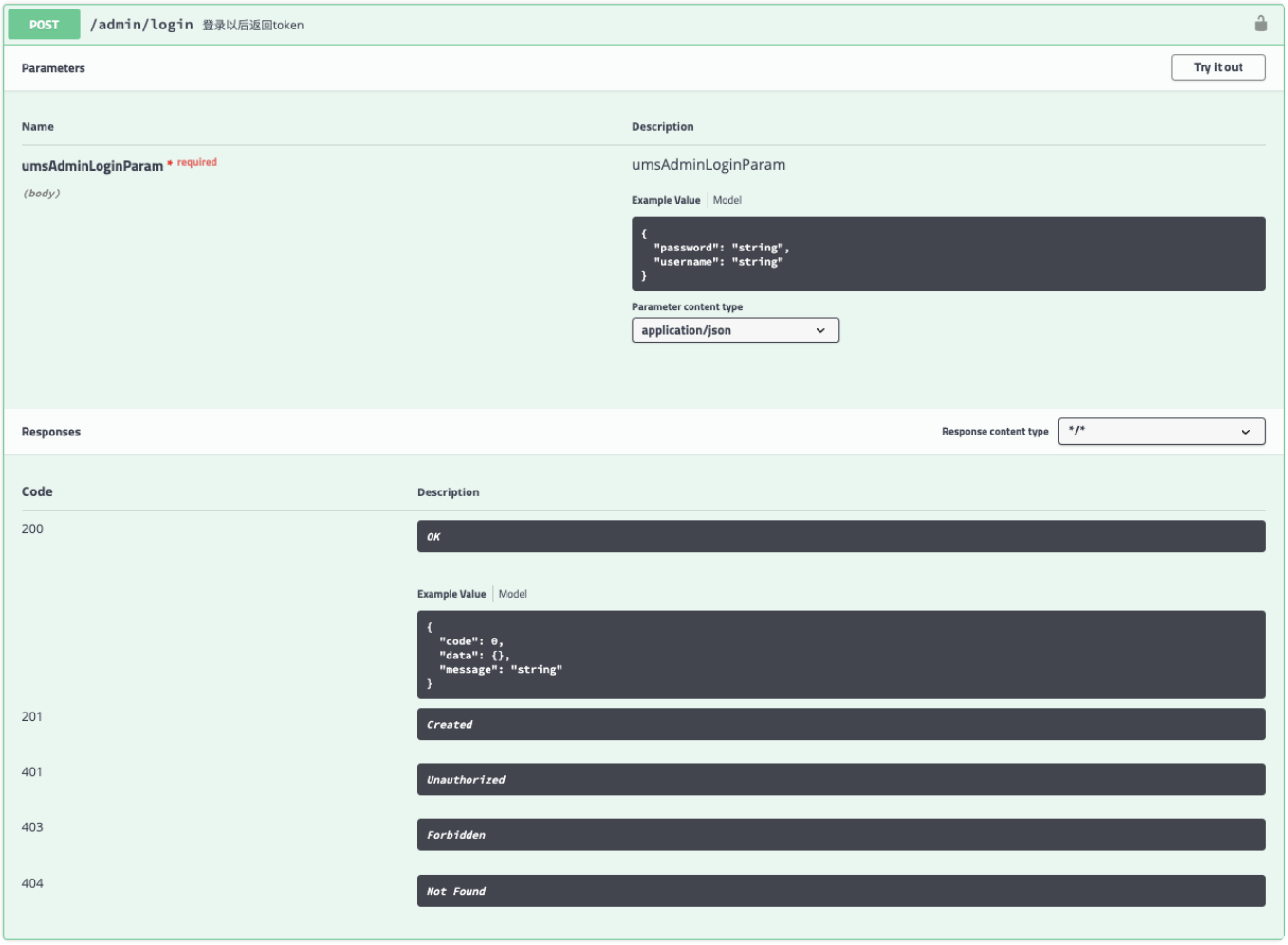
## 如何实现回放关联？

这里我们引入“会员登录”和“查询所有后台资源分类”两个接口为例。

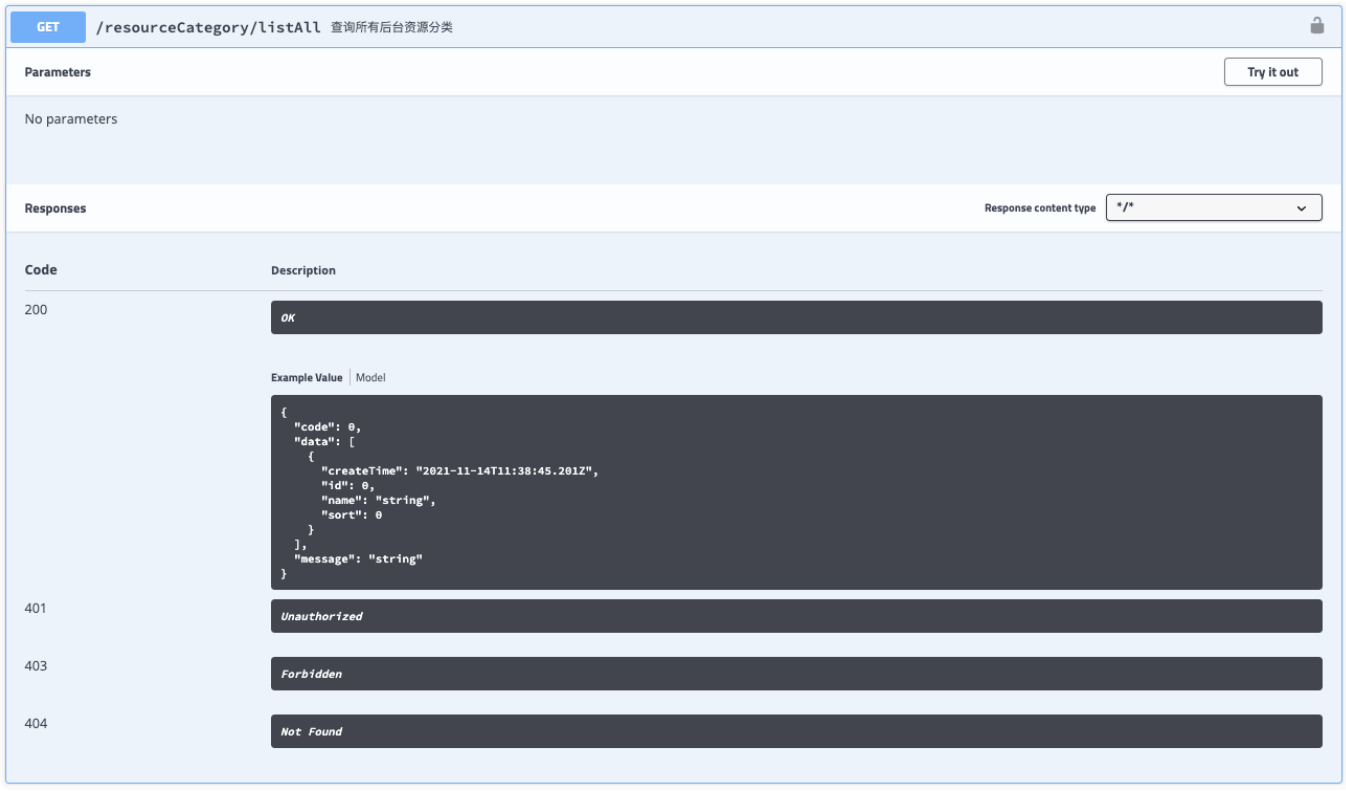
你可以先看看这张整体的请求交互示意图：


客户端
服务端
1
1
POST /admin/login
Content-Type: application/json
Authorization: true
{
"password": "123456",
"username": "test"
}
{
"code": 200,
"message": "操作成功",
"data": {
"token": "xxx"
}
}
2
2
GET /resource/listAll
token: xxx
{
"code": 200,
"message": "操作成功",
"data": [
{
"id": 1,
"name": "商品品牌管理",
"url": "/brand/**"
},
....
}
3
极客时间

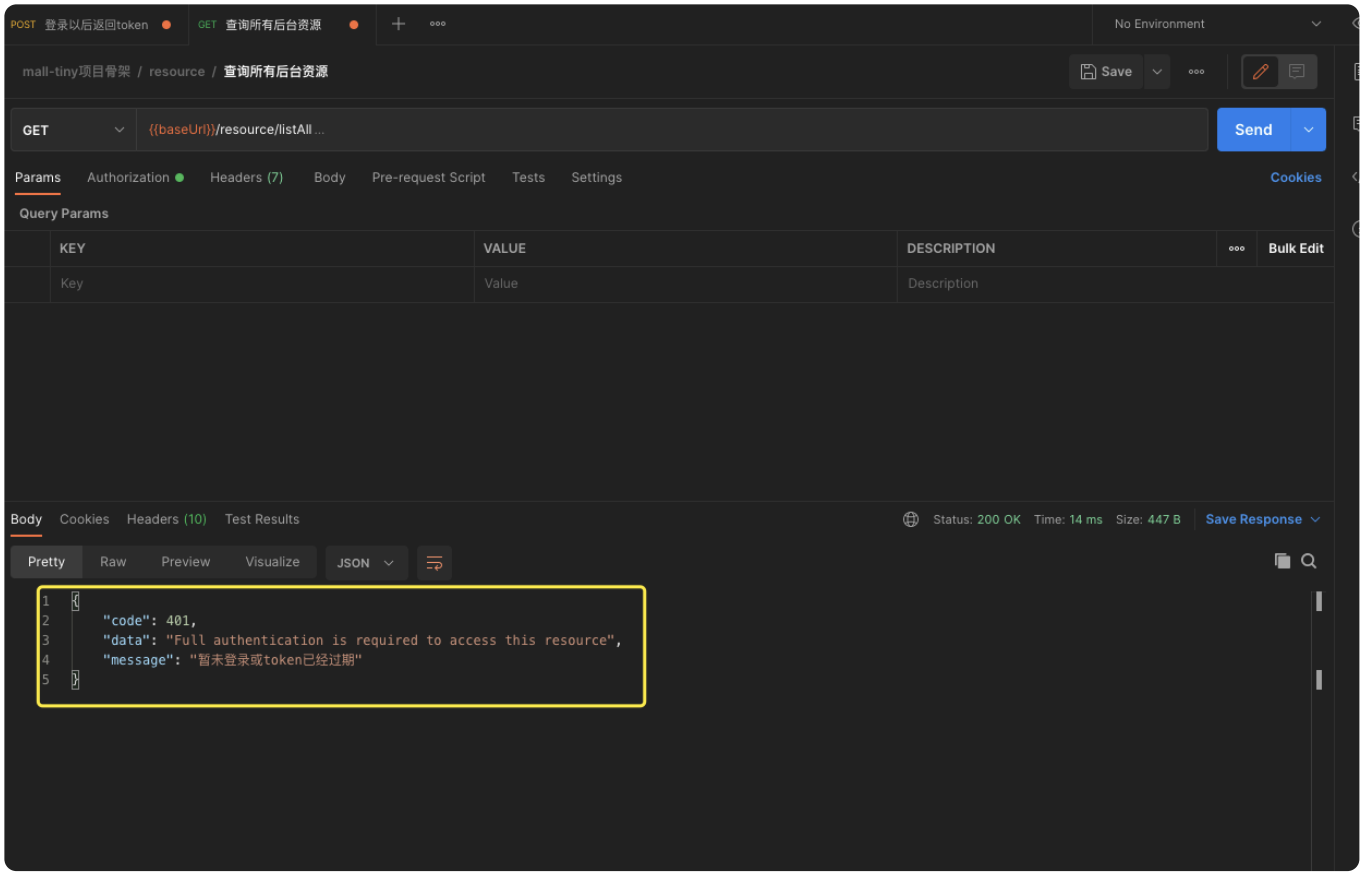
会员登录




查询所有后台资源分类




我们知道 token 是有时效的，如果失效，那么二次请求服务端校验就会失败。如下图：




下面我们具体来演示下如何解决 token 关联的问题。

第一步，创建一个流量录制的命令：

复制代码

```
#!/bin/bash

PORT="8081"
OUT_FILE="request.gor"

sudo ./goreplay --input-raw :$PORT --output-file=$OUT_FILE  -output-file-appen
```

录制下的流量文件如下：

复制代码

```
1 d1ae1f9100000001e404ee86 1635588156669182000 0
POST /admin/login HTTP/1.1
Content-Type: application/json
Authorization: true
User-Agent: PostmanRuntime/7.28.4
Accept: */*
Postman-Token: 480f15ca-53df-44fd-8980-5e9118b2107e
Host: localhost:8081
```

```
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 52

{
    "password": "123456",
    "username": "test"
}

🙈🙉🙊
2 d1ae1f9100000001e404ee86 1635588156828973000 458000
HTTP/1.1 200
Content-Length: 254
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json
Date: Sat, 30 Oct 2021 10:02:36 GMT
Keep-Alive: timeout=60
Connection: keep-alive

{"code":200,"message":"操作成功","data":{"tokenHead":"","token":"eyJhbGciOiJIUz

🙈🙉🙊
1 d1ae1f9100000001e405029a 1635588192031592000 0
GET /resource/listAll HTTP/1.1
token: eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ0ZXN0IiwiY3JlYXRlZCI6MTYzNTU4ODE1Njc5NC
User-Agent: PostmanRuntime/7.28.4
Accept: */*
Postman-Token: 4bc2152e-dbd4-4b2a-b880-72ed5ee4303a
Host: localhost:8081
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

🙈🙉🙊
2 d1ae1f9100000001e405029a 1635588192064563000 1084000
HTTP/1.1 200
Content-Length: 3997
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

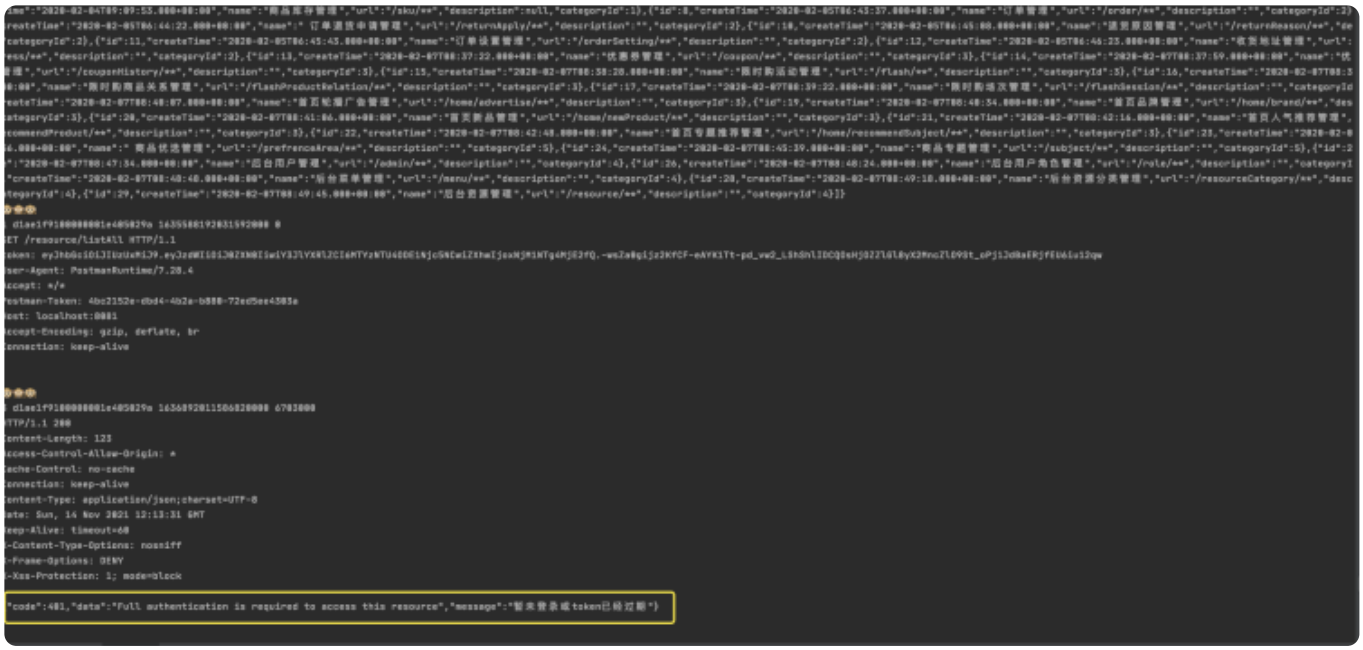
```
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json
Date: Sat, 30 Oct 2021 10:03:12 GMT
Keep-Alive: timeout=60
Connection: keep-alive

{"code":200,"message":"操作成功","data":[{"id":1,"createTime":"2020-02-04T09:04
```

第二步，创建一个流量回放 Shell 脚本。

复制代码

```
#!/bin/bash
## Usage: ./replay.sh

OUTPUT="http://127.0.0.1:8081"
INPUT_FILE="requests.gor"

sudo ./goreplay --input-file $INPUT_FILE --input-file-loop --output-http=$OUTP
```

第三步，我们尝试进行一次回放操作。

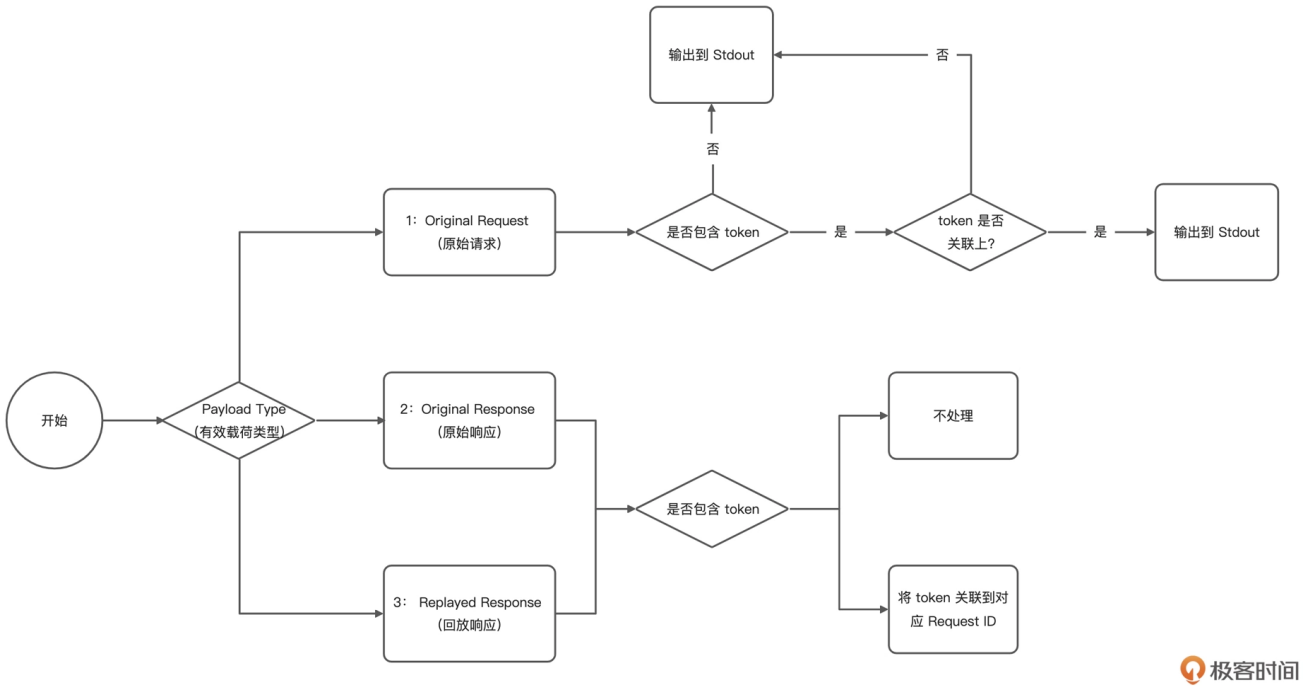


等待一会，我们看到回放的【查询所有后台资源分类】接口已经失败了，提示 token 失效了。

要怎么解决这个问题呢？

在这种情况下，我们需要实时将来自录制的 token 关联到来自回放响应的 token 上 ，然后使用关联的 token 修改回放的请求。我们使用 GoReplay 存储库中的这个方便的示例进行扩展。

所涉及的基本算法你可以看看下面这张图片。




因为原始服务器没有预定义的 token，而回放服务器有自己的 token，它不能与原始服务器同步。所以不使用中间件或者中间件只使用请求有效 payload，都会使得 token 失效。

为了解决这个问题，我们的中间件应该考虑回放和源服务器的响应，存储’ originalToken -> replayedToken '别名，并使用此 token 重写所有请求以使用回放别名。

顺着这个思路，我们看下第四步，创建 token 关联中间件程序。

复制代码

```
1 package main
2
3 import (
4   "bufio"
5   "bytes"
```

```
  "encoding/hex"
  "fmt"
  "github.com/bitly/go-simplejson"
  "github.com/buger/goreplay/proto"
  "os"
)

// requestID -> originalToken
// 请求 ID -> 原始 Token
var originalTokens map[string][]byte

// originalToken -> replayedToken
// 原始 Token -> 回放 Token
var tokenAliases map[string][]byte

var json_data interface{}



func main() {
  originalTokens = make(map[string][]byte)
  tokenAliases = make(map[string][]byte)

  scanner := bufio.NewScanner(os.Stdin)

  for scanner.Scan() {
    encoded := scanner.Bytes()
    buf := make([]byte, len(encoded)/2)
    hex.Decode(buf, encoded)

    process(buf)
  }
}

func process(buf []byte) {
  // First byte indicate payload type, possible values:
  //  1 - Request
  //  2 - Response
  //  3 - ReplayedResponse
  // 第一个字节表示有效负载类型，可能的值：
  // 1 - 请求
  // 2 - 响应
  // 3 - 回放响应
  payloadType := buf[0]
  headerSize := bytes.IndexByte(buf, '\n') + 1
  header := buf[:headerSize-1]

  // Header contains space separated values of: request type, request id, and
  // Header 包含空格分隔的值:请求类型，请求 id，请求开始时间(或响应的往返时间)
  meta := bytes.Split(header, []byte(" "))

    // For each request you should receive 3 payloads (request, response, repl
```

```
// 对于每个请求，你应该收到 3 个有效负载(request, response, replayed response)，具
reqID := string(meta[1])
payload := buf[headerSize:]

Debug("Received payload:", string(buf))

switch payloadType {
case '1': // Request
  if bytes.Equal(proto.Path(payload), []byte("/admin/login")) {
    originalTokens[reqID] = []byte{}
    Debug("Found token request:", reqID)
  } else {
    //token, vs, _ := proto.PathParam(payload, []byte("token")) //取到回放响应
    token := proto.Header(payload, []byte("token")) //取到原始的 token 值

    Debug("Received token:", string(token))

    if len(token) != 0 { // If there is GET token param
      Debug("If there is GET token param")
      Debug("tokenAliases", tokenAliases)
      if alias, ok := tokenAliases[string(token)]; ok {     //检查要替换的 tok
        Debug("Received alias")
        // Rewrite original token to alias
        payload = proto.SetHeader(payload, []byte("token"), alias)  //将原始的
        // Copy modified payload to our buffer
        buf = append(buf[:headerSize], payload...)
      }
    }
  }

  // Emitting data back
  os.Stdout.Write(encode(buf)) //重写请求准备发往回放服务
case '2': // Original response
  if _, ok := originalTokens[reqID]; ok {
    jsonObject, err := simplejson.NewJson([]byte(proto.Body(payload)))
    if err != nil {
      fmt.Println(err)
    }

    result := jsonObject.Get("data")
    token := result.Get("token")
    secureToken:=token

    f ,_:=secureToken.Bytes()

    originalTokens[reqID] = f
    Debug("Remember origial token:", f)

  }
case '3': // Replayed response
  if originalToken, ok := originalTokens[reqID]; ok {
```

```
        delete(originalTokens, reqID)



        jsonObject, err := simplejson.NewJson([]byte(proto.Body(payload)))
        if err != nil {
          fmt.Println(err)
        }
        result := jsonObject.Get("data")
        token := result.Get("token")
        f ,_:=token.Bytes()
        tokenAliases[string(originalToken)] = f //拿到现在的 token 值用来替换掉过去的

        Debug("Create alias for new token token, was:", string(originalToken), "
      }
    }
}

func encode(buf []byte) []byte {
  dst := make([]byte, len(buf)*2+1)
  hex.Encode(dst, buf)
  dst[len(dst)-1] = '\n'

  return dst
}

func Debug(args ...interface{}) {
  if os.Getenv("GOR_TEST") == "" { // if we are not testing
    fmt.Fprint(os.Stderr, "[DEBUG][TOKEN-MOD] ")
    fmt.Fprintln(os.Stderr, args...)
  }

}
```

我们可以使用 process 函数异步处理原始请求或回放响应从而重新设置 token。由于 GoReplay 的每个三元组（请求、响应、回放响应）共享一个请求 ID，因此到达中间件的第一个响应可以将它的 token 关联到请求 ID。当第二个响应到达时，我们就可以访问两个 token 了。我们可以将原始 token 关联到回放的 token，并能够一一对应（因为第二个响应类型也可用）。

好了，这样中间件就写完了，我们一起来测试一下。

我们先创建一个运行中间件的 Shell 脚本 middleware_wrapper.sh。

复制代码

```
#!/bin/bash
go run token_modifier.go
```

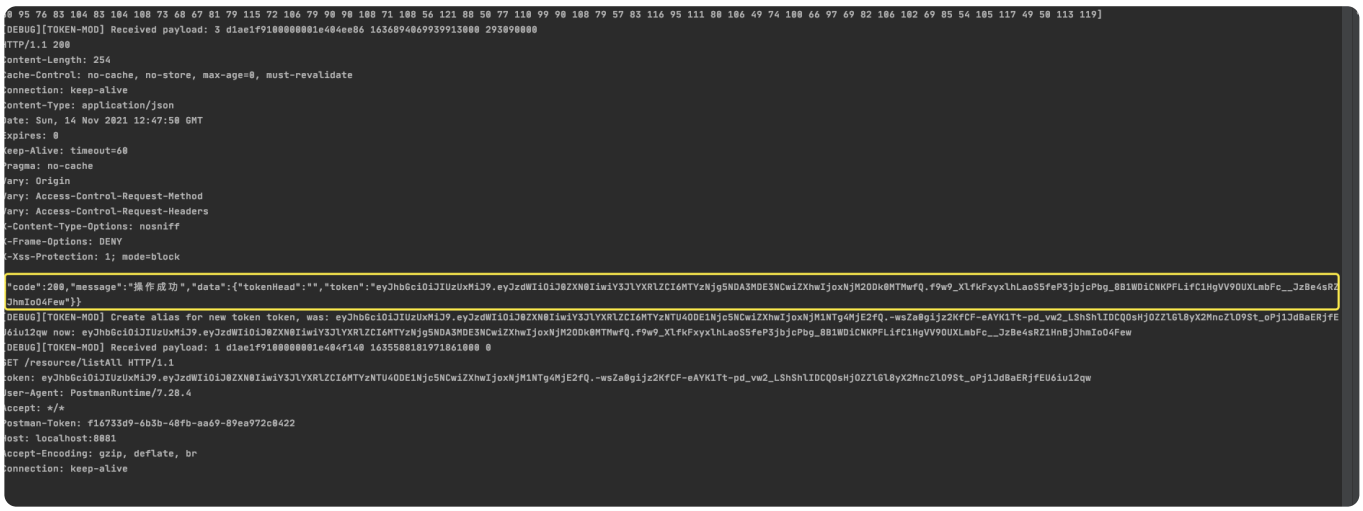
第二步，修改启动回放的 Shell 脚本 replay.sh。

复制代码

```
#!/bin/bash
## Usage: ./replay.sh

OUTPUT="http://127.0.0.1:8081"
MIDDLEWARE="./middleware_wrapper.sh"
INPUT_FILE="requests.gor"

sudo ./goreplay --input-file $INPUT_FILE --input-file-loop --output-http=$OUTP
```

最后一步，我们就要紧盯运行控制台了。

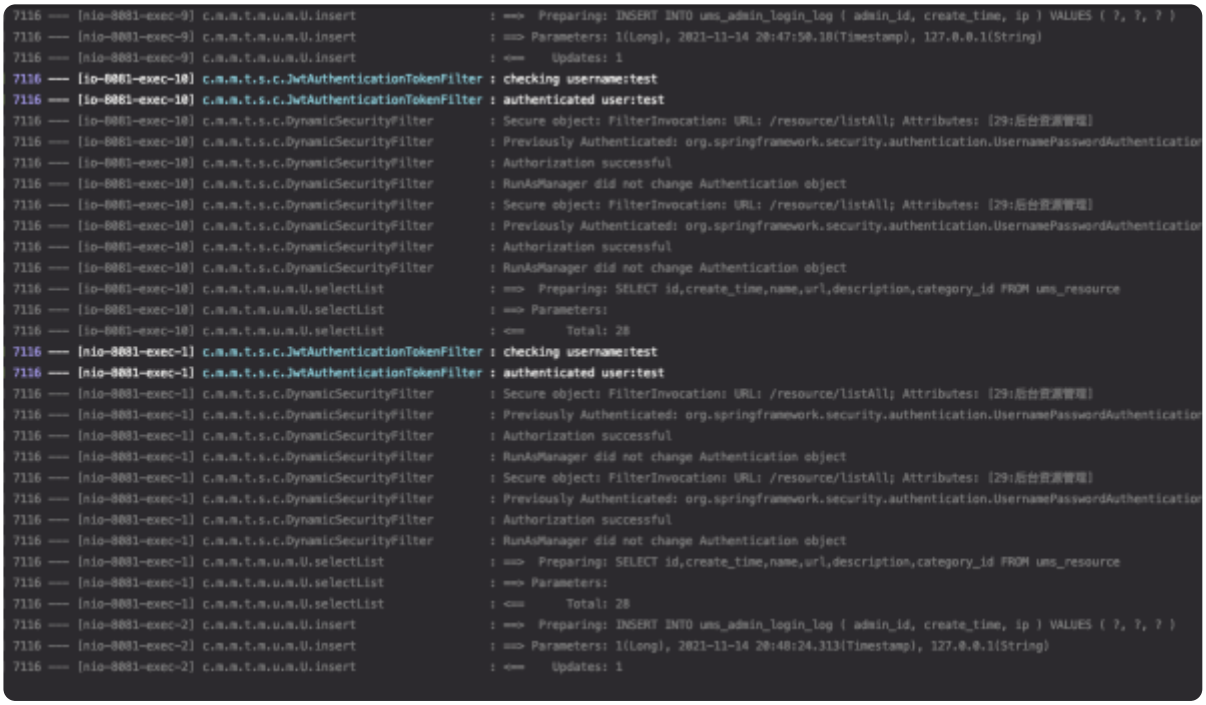
登录接口实时返回的 token。




【查询所有后台资源分类】接口，可以看到已经成功替换到回放响应的 token 了。

【查询所有后台资源分类】接口，回放响应的数据也是正常的。

服务端日志显示正常。




好了，到这里，我们的动态数据关联功能就已经实现了。

## 总结

好了，这节课就讲到这里。刚才，我们一起梳理了关联的基本概念、 GoReplay 中间件（Middleware ）原理、常用的用法。我们还通过例子演示了 GoReplay 如何通过扩展 Middleware 做到关联功能。实际上，我们可以在中间件上实现任何自定义逻辑，比如认证、复杂的重写和筛选请求等。

下一节课，我们将进入具体的分布式改造环节，我会通过案例演示如何做分布式平台改造工作。

## 课后题

学完这节课，请你思考两个问题：

1. 你有没有使用过 Middleware，谈谈你对 Middleware 应用的一些心得吧！
2. 相比 JMeter，你觉得 GoReplay 关联的难度在什么地方？

欢迎你在留言区与我交流讨论。当然了，你也可以把这节课分享给你身边的朋友，他们的一些想法或许会让你有更大的收获。我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | 压测平台：高效搭建 GoReplay 压测平台

下一篇 23 | 压测平台：如何改造对象存储和性能监控？

更多学习推荐

## 精选留言 (1)

写留言

**aMaMiMou**
2021-12-09

老师，您好：
如果是通过goreplay或者tcpcopy在线引流的方式直接把捕获的流量打到目标待压测端，是否就不再需要处理中间态的关联动态数据了呢？因为实时捕获的请求里是包含了服务端响应的动态数据包。

展开

作者回复: 实时引流可以不用处理，但录制回放就需要考虑了。